C++ STL best and worst performance features and how to learn from them

Danila Kutenin
Google
kutdanila@gmail.com
Telegram: @Danlark
Twitter: @Danlark1

About myself

- Worked at Yandex on core search engines
- Working at Google on distributed data processing
- Primary expertise is C/C++, low-level design, distributed systems design

Agenda

- Is C++ about performance?
- Performance problems
  - ABI
  - Compiler
  - Algorithmic
- STL performance experience

C++ performance

C++ is performant. Because of philosophy [1].

- What you don’t use, you don’t pay for
- What you do use, you couldn’t hand-code any better

[1] B. Stroustrup
https://dl.acm.org/doi/abs/10.1145/3386320

C++ performance

Chandler Carruth “There Are No Zero-cost Abstractions”

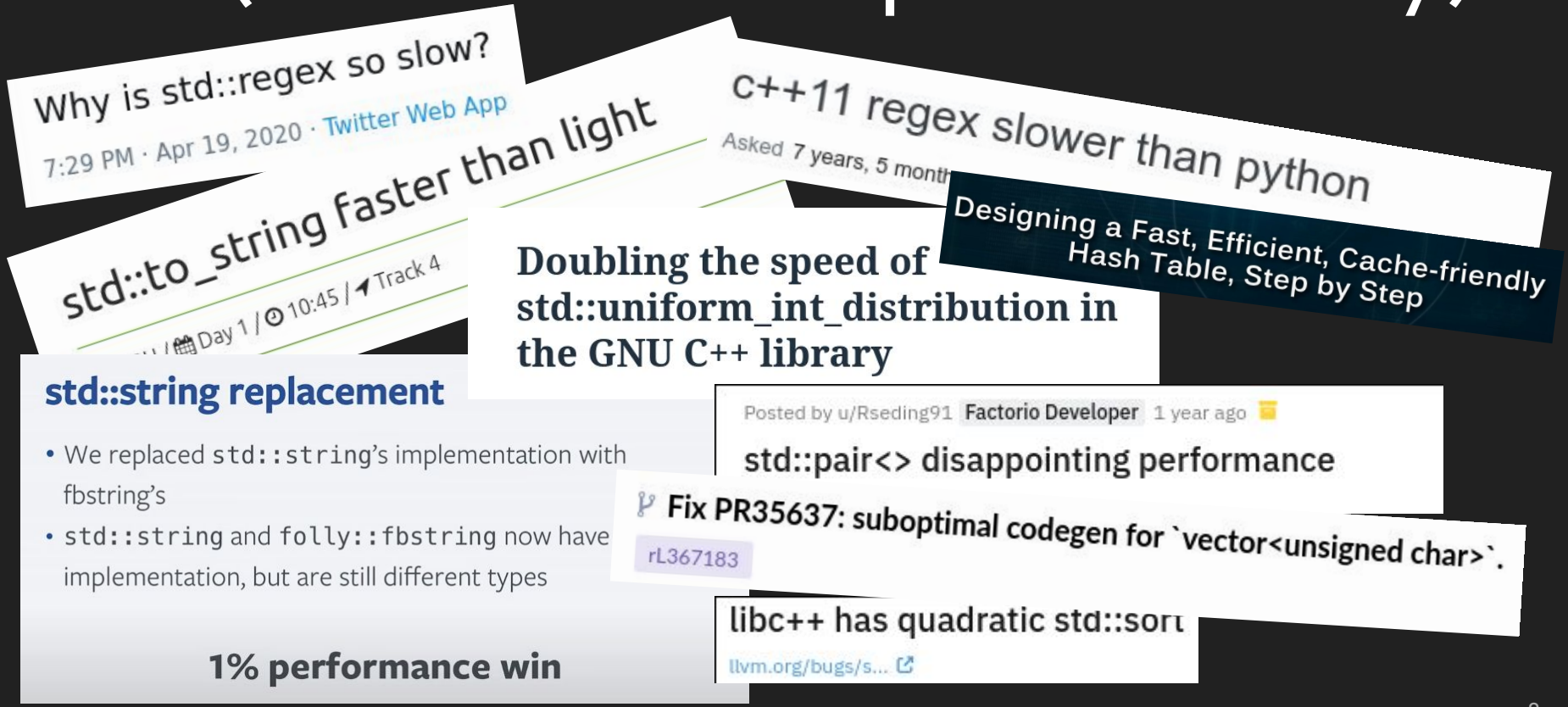
STL (Standard Template Library)

- Provides “standard” things. `std::vector`
- Hard to correctly implement “convenient” things. `std::shared_ptr`

STL (Standard Template Library)

Is it actually performant?

STL (Standard Template Library)



STL (Standard Template Library)

- Results in self made libraries
  - Abseil
  - Folly
  - EASTL
- Why?
  - ABI compatibility and faster progress
  - Speed is simply money

Outstanding Types/Containers

- `std::array`
- `std::optional, std::variant`
- `std::atomic`
- `std::span, std::string_view`
- `<algorithm>`

Encouraged to use in many places

Debatable Types/Containers

- `std::vector`
- `std::string`
- `std::set, std::map`

People still debate. Readability costs outweigh the last percentages

Bad Types/Containers

- `std::pair, std::tuple`
- `std::unordered_*`
- `std::regex`

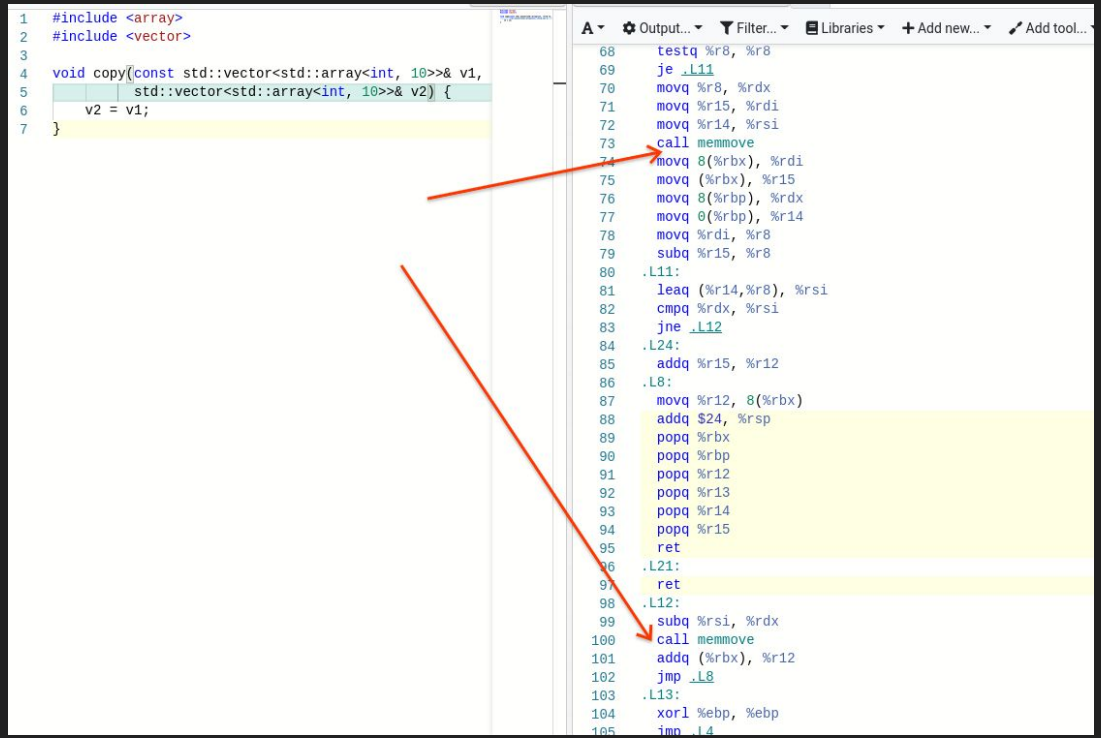
Last two are banned in many places

std::array<T>

- No constructors, copy operators, destructors
  - Rule of zero is the key to success
- The performance is as `T[N]` with the convenient helper functions

std::array<T>

std::is_trivially_copyable if T is



Trick #1

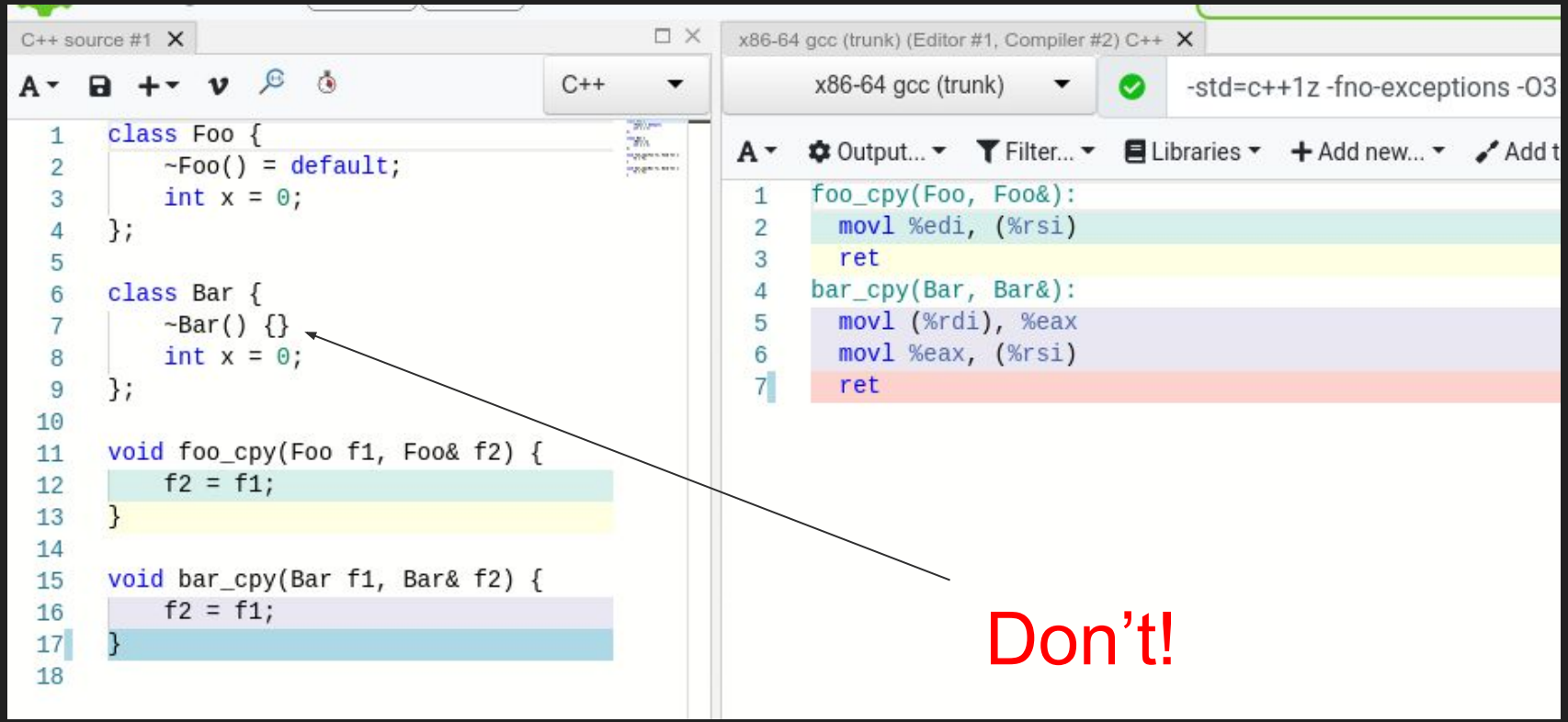
- If possible, make your type trivial
  - Trivially destructible types
    - It allows to “reuse” the object
  - Trivially copyable types can be `memcpy`’ed
    - `mem*` are highly platform optimized

Trick #1

The SysV ABI specification, section 3.2.3 Parameter Passing says:

If a C++ object has either a non-trivial copy constructor or a non-trivial destructor, it is passed by invisible reference.

Trick #1


C++ source #1
C++
1 class Foo {
2 ~Foo() = default;
3 int x = 0;
4 };
5
6 class Bar {
7 ~Bar() {}
8 int x = 0;
9 };
10
11 void foo_cpy(Foo f1, Foo& f2) {
12 f2 = f1;
13 }
14
15 void bar_cpy(Bar f1, Bar& f2) {
16 f2 = f1;
17 }
18
x86-64 gcc (trunk) (Editor #1, Compiler #2) C++
x86-64 gcc (trunk)
-std=c++1z -fno-exceptions -O3
Output...
Filter...
Libraries
Add new...
1 foo_cpy(Foo, Foo&):
2 movl %edi, (%rsi)
3 ret
4 bar_cpy(Bar, Bar&):
5 movl (%rdi), %eax
6 movl %eax, (%rsi)
7 ret
Don’t!

std::optional<T>

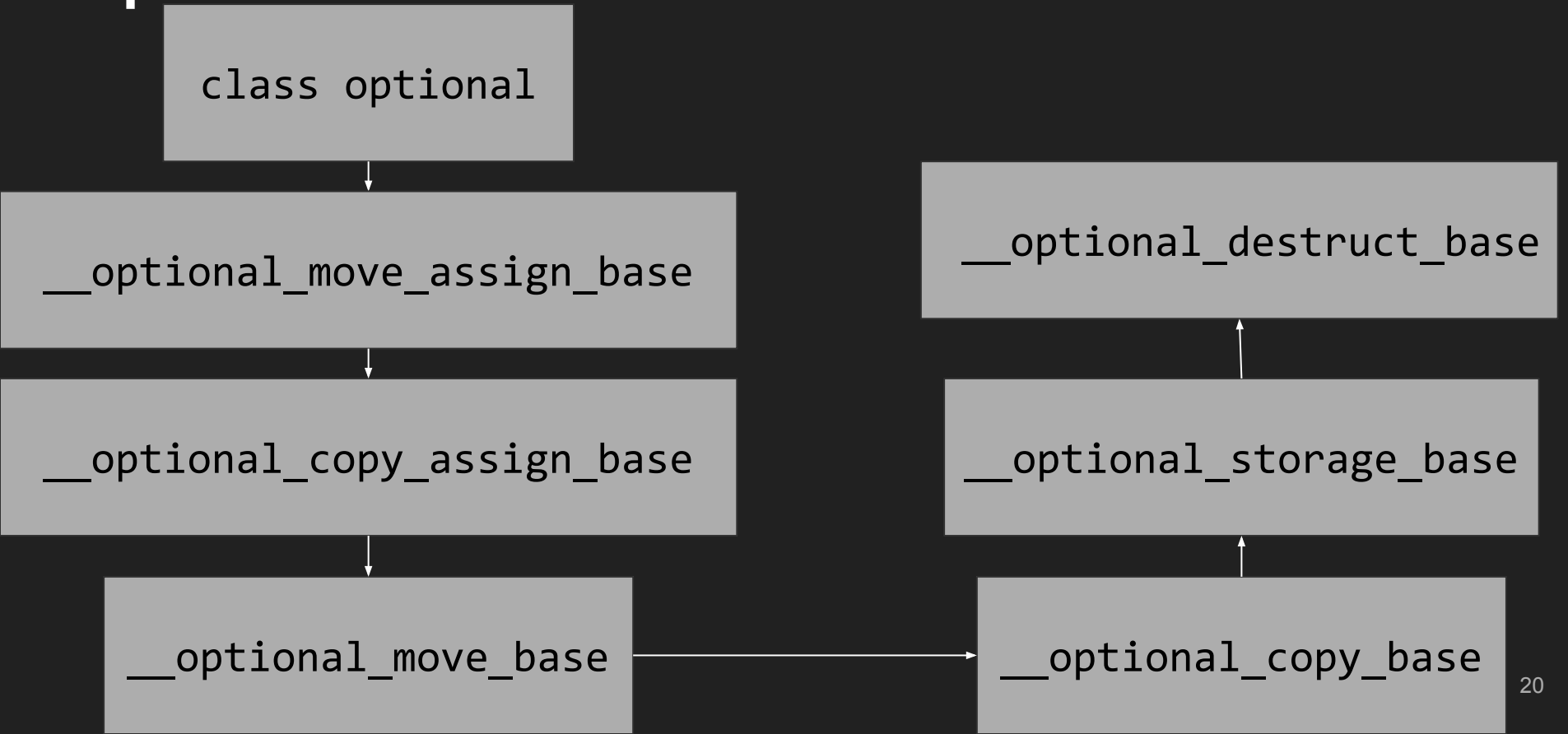
std::optional<T>

[1]
~optional();

1. *#Effects: If is_trivially_destructible_v<T> != true and *this contains a value, calls val->T::~T()*
2. *#Remarks: If is_trivially_destructible_v<T> is true, then this destructor is **trivial**.*

[1] utilities.optional.destructor

Optional Perf At What Cost?



Optional Perf At What Cost?

`__optional_destruct_base`

Why?
Partial specializations/SFINAE on special member functions are forbidden

Optional Perf At What Cost?

1420 Lines of Code*

*libc++ implementation

Optional Perf At What Cost?

P0602R4

variant and optional should propagate copy/move triviality

Why is the construction of std::optional<int> more expensive than a std::pair<int, bool>?

Fixed in libstdc++8

Evil side. `std::pair`, `std::tuple`

```cpp
#include <utility>
#include <vector>

struct MyPair {
    int a = 0;
    int b = 0;
};

static void CopyMyPair(benchmark::State& state) {
  std::vector<MyPair> v1(state.range(0));
  std::vector<MyPair> v2;
  for (auto _ : state) {
    v2 = v1;
    benchmark::DoNotOptimize(v2);
  }
}
BENCHMARK(CopyMyPair)->Arg(100000);

static void CopyStdPair(benchmark::State& state) {
  std::vector<std::pair<int, int>> v1(state.range(0));
  std::vector<std::pair<int, int>> v2;
  for (auto _ : state) {
    v2 = v1;
    benchmark::DoNotOptimize(v2);
  }
}
BENCHMARK(CopyStdPair)->Arg(100000);
```

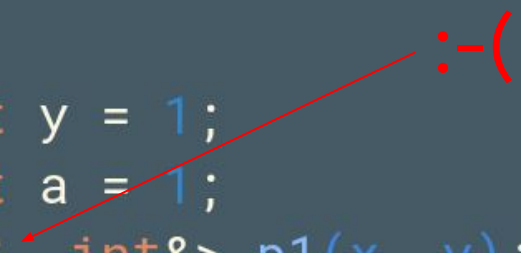
What?

```
pair& operator=(typename conditional<
                is_copy_assignable<first_type>::value &&
                is_copy_assignable<second_type>::value,
            pair, __nat>::type const& __p)
{
  first = __p.first;
  second = __p.second;
  return *this;
}
```

Why?

```cpp
#include <iostream>
#include <utility>

int main() {
  int x = 0; int y = 1;
  int z = 0; int a = 1;
  std::pair<int&, int&> p1(x, y);
  std::pair<int&, int&> p2(a, z);
  p1 = p2;
  std::cout << x << ' ' << y << std::endl;
  return 0;
}
```



Why?

Same with tuple. `std::tie` works the same way

Why?

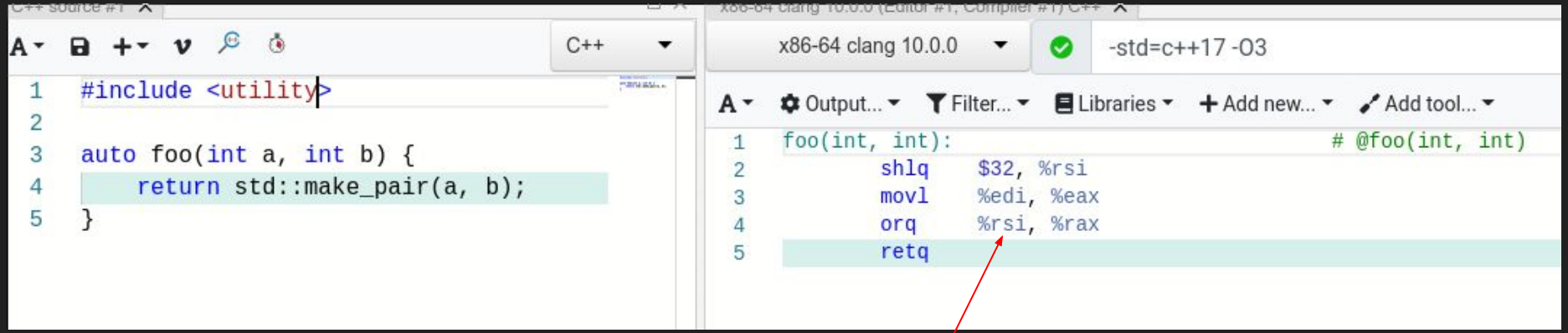
```
template <class ...Tp>
inline tuple<Tp&...> tie(Tp&... t) noexcept {
    return tuple<Tp&...>(t...);
}
```

Why?

A defaulted copy assignment operator for class T is defined as deleted if any of the following is true:

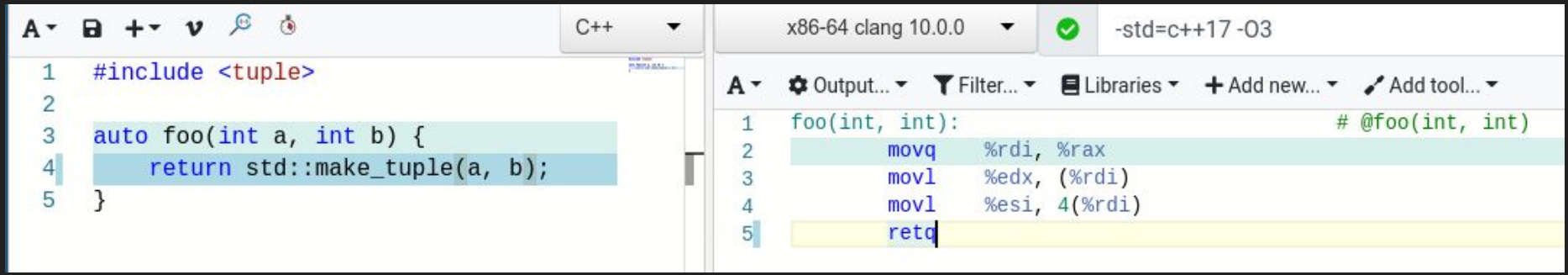
- ...
- T has a non-static data member of a reference type;
- ...

Good news?



Combines in 1 register

Good news? Not for tuple



Bad news?

ABI break

What is ABI?

Translation unit interactions. Including:

- The mangled name for a C++ function
- The mangled name for a type, including templates.
- The number of bytes (sizeof) and the alignment
- The semantics of the bytes in the binary representation of an object.
- Register-level calling conventions.

What is ABI?

```
struct Example1 {
  int a = 0;
  int b = 0;
  Example1& operator=(const Example1& other) {
      a = other.a;
      b = other.b;
      return *this;
  }
};

struct Example2 {
  int a = 0;
  int b = 0;
  Example2& operator=(const Example2& other) = default;
};
```

What is ABI?

```
void Example1Copy(const Example1& e,
                  Example1& e_other) {
  e_other = e;
}

void Example2Copy(const Example2& e,
                  Example2& e_other) {
  e_other = e;
}
```

What is ABI?

```
^^>>> g++ -std=gnu++17 -O0 test_example.cpp -o test_example
^^>>> nm -C test_example | tail
0000000000001200 T __libc_csu_fini
0000000000011a0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
0000000000001168 T main
0000000000010a0 t register_tm_clones
0000000000001040 T _start
0000000000004028 D __TMC_END__
0000000000001125 T Example1Copy(Example1 const&, Example1&)
000000000000114b T Example2Copy(Example2 const&, Example2&)
0000000000001174 W Example1::operator=(Example1 const&)
^^>>>
```

Future?

P0848R3

Conditionally Trivial Special Member Functions

```
template <typename T>
concept C = /* ... */;

template <typename T>
struct X {
    // #1
    X(X const&) requires C<T> = default;

    // #2
    X(X const& ) { /* ... */ }
};
```

Optional implementation down to 390 LOC

Trick #2

Write =default in your code.
Always when possible.

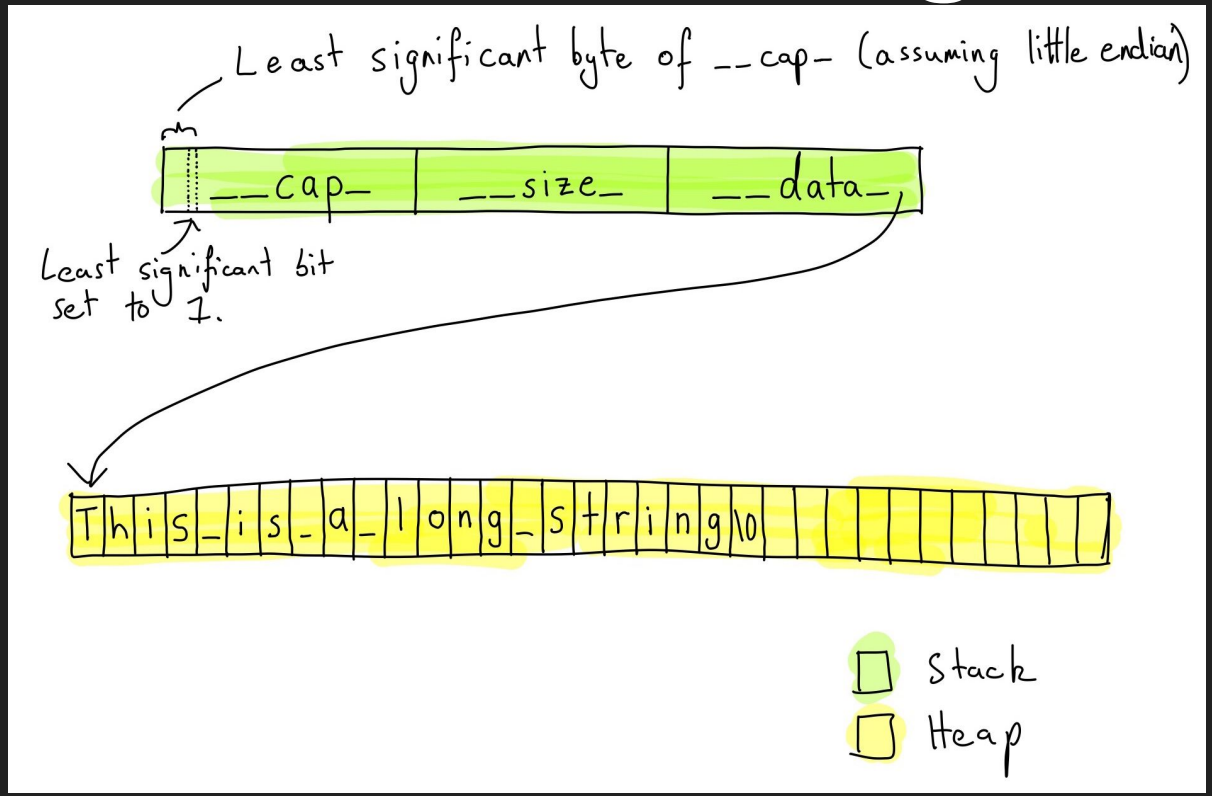
std::string

- Small/Short string optimization
  - We must store pointer, size and capacity
  - Reuse these bytes when the string is small
- Dates back to 2000-2001

std::string

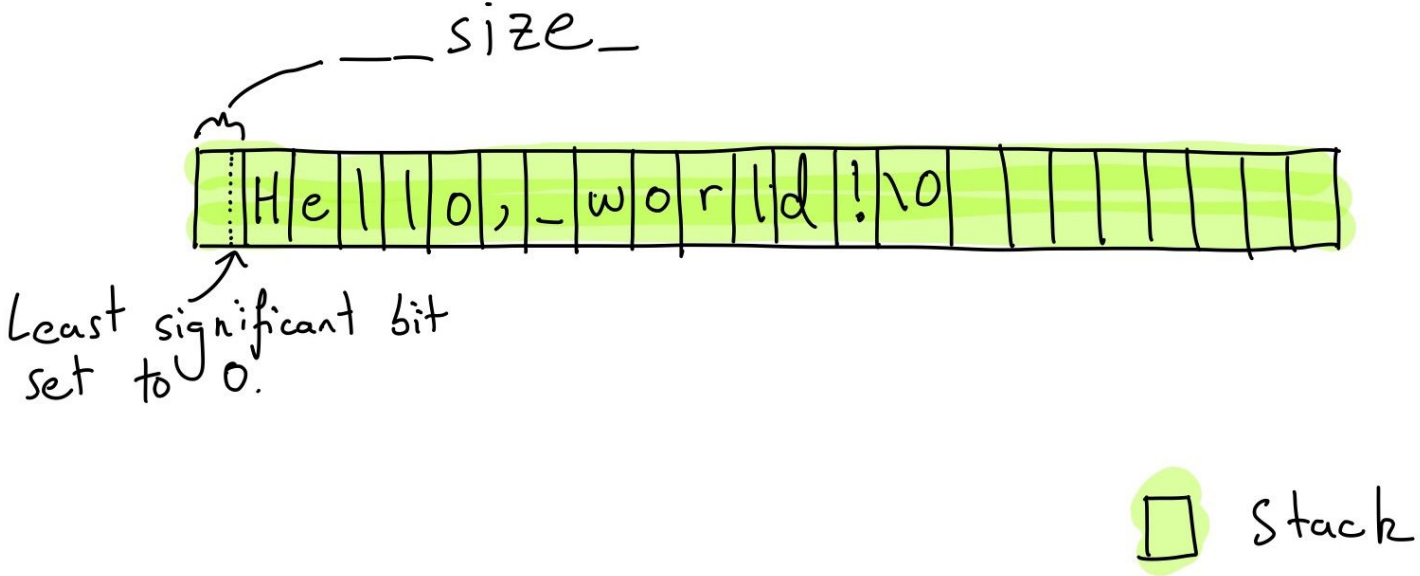
- libc++: 22 bytes
- libstdc++: 15 bytes
- MSVC STL: 15 bytes
- FBString: 23 bytes: https://www.youtube.com/watch?v=kPR8h4-qZdk
- Yandex: 0 bytes, fully COW

std::string



https://joellaity.com/2020/01/31/string.html

std::string



https://joellaity.com/2020/01/31/string.html

std::string

- `std::function` uses the same technique
- The trick can be useful for highly accessed data

- `` `const std::string&` `` should almost die
  - Use `std::string_view`, two registers, no indirection, cheap copy, pass by value

std::string

```cpp
#include <string>
#include <string_view>

size_t size_string(const std::string& s) {
    return s.size();
}

size_t size_string_view(std::string_view sv) {
    return sv.size();
}
```

```
size_string(std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::al
        movzbl  (%rdi), %eax
        testb   $1, %al
        je      .LBB0_1
        movq    8(%rdi), %rax
        retq
.LBB0_1:
        shrq    %rax
        retq
size_string_view(std::__1::basic_string_view<char, std::__1::char_traits<char> >):
        movq    %rsi, %rax
        retq
```

Check for small string

std::function

```
std::function<void()> = [&]() {
   //...
};
```

Trick #3

Use `std::string_view` and `std::span` almost everywhere

Trick #4

Remember about small object optimizations, e.g. don’t capture blindly by reference in lambda

std::string

```
std::vector<std::string> to_join;
std::string result;
for (const auto& part : to_join) {
  result += part;
}
```

std::string

- absl::StrJoin, folly::join
  - Sums all sizes, does 1 allocation

```
template <typename string_type>
struct ResizeUninitializedTraits<
    string_type,
    absl::void_t<decltype(std::declval<string_type&>()
                          .__resize_default_init(237))>>
{
  using HasMember = std::true_type;
  static void Resize(string_type* s, size_t new_size) {
    s->__resize_default_init(new_size);
  }
};
```

Trick #5

As of C++20, write your own string operations library or use the existing external one

`std::unordered_*`

- Check if you need pointer stability (likely not)
- C++17 still does not support heterogeneous lookups
  - Many other libraries do, for example, `absl::flat_hash_*`
- You can outperform std:: by 10-20x

https://github.com/google/hashtable-benchmarks

Trick #6

If you have enough hash table usages, use external ones or even write your own

<algorithm>

- Use them, they don’t have ABI problems
  - They are constantly optimized in libraries
  - Compilers produce better SIMD code with time
  - Only several are still debatable
    - E.g. `std::sort`, `std::nth_element`
    - Still use them

std::rotate

- Two algorithms, rotating by k where k < n.
  - GCD rotate. Moves n + gcd(n, k) times.
    - Requires random access
  - Forward rotate. Moves between 3/2n and 3n times.
    - Can be done with forward iterators

std::rotate

```
template <class ForwardIterator>
inline ForwardIterator
rotate(ForwardIterator first, ForwardIterator middle,
       ForwardIterator last) {
    return rotate(first, middle, last,
                  typename iterator_traits<ForwardIterator>
                      ::iterator_category());
}
```

std::rotate

```
template <class RandomAccessIterator>
inline RandomAccessIterator
rotate(RandomAccessIterator first, RandomAccessIterator middle,
       RandomAccessIterator last, random_access_iterator_tag) {
    if (is_trivially_move_assignable<value_type>::value) {
        if (next(first) == middle)
            return rotate_left(first, last);
        if (next(middle) == last)
            return rotate_right(first, last);
        return rotate_gcd(first, middle, last);
    }
    return rotate_forward(first, middle, last);
}
```

std::rotate

```
template <class ForwardIterator>
inline ForwardIterator
rotate(ForwardIterator first, ForwardIterator middle,
       ForwardIterator last, forward_iterator_tag) {
  if (is_trivially_move_assignable<value_type>::value) {
    if (next(first) == middle)
      return rotate_left(first, last);
  }
  return rotate_forward(first, middle, last);
}
```

std::copy

```
template <class InputIterator, class OutputIterator>
inline OutputIterator
copy(InputIterator first, InputIterator last,
     OutputIterator result) {
  return __copy(first, last, result);
}
```

std::copy

```
template <class InputIterator, class OutputIterator>
inline OutputIterator
copy(InputIterator first, InputIterator last,
     OutputIterator result) {
  return __copy(first, last, result);
}
```

std::copy

```
template <class _Tp, class _Up>
inline
typename enable_if
<
    is_same<typename remove_const<_Tp>::type, _Up>::value &&
    is_trivially_copy_assignable<_Up>::value,
    _Up*
>::type
__copy(_Tp* __first, _Tp* __last, _Up* __result)
{
    const size_t __n = static_cast<size_t>(__last - __first);
    if (__n > 0)
        _VSTD::memmove(__result, __first, __n * sizeof(_Up));
    return __result + __n;
}
```

Bug?

std::reverse

```
void reverse(const _BidIt _First, const _BidIt _Last) {
  if constexpr (_Allow_vectorization
                && sizeof(_Elem) == 1) {
    __std_reverse_trivially_swappable_1(_UFirst, _ULast);
    return;
  } else if constexpr (_Allow_vectorization
                       && sizeof(_Elem) == 2) {
    __std_reverse_trivially_swappable_2(_UFirst, _ULast);
    return;
  }
  ...
}
```

std::reverse

```
constexpr bool _Allow_vectorization =
        conjunction_v<is_pointer<decltype(_UFirst)>,
                      _Is_trivially_swappable<_Elem>,
                      negation<is_volatile<_Elem>>>;
```

std::reverse

```
void  __std_reverse_trivially_swappable_2(void* _First, void* _Last) noexcept {
  if (_Byte_length(_First, _Last) >= 64 &&
      _bittest(&__isa_enabled, __ISA_AVAILABLE_AVX2)) {
    const __m256i _Reverse_short_lanes_avx = _mm256_set_epi8( //
        1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14, //
        1, 0, 3, 2, 5, 4, 7, 6, 9, 8, 11, 10, 13, 12, 15, 14);
    void* _Stop_at                          = _First;
    _Advance_bytes(_Stop_at, _Byte_length(_First, _Last) >> 6 << 5);
  do {
    _Advance_bytes(_Last, -32);
    const __m256i _Left  = _mm256_permute4x64_epi64(
                           _mm256_loadu_si256(static_cast<__m256i*>(_First)), 78);
    const __m256i _Right = _mm256_permute4x64_epi64(
                           _mm256_loadu_si256(static_cast<__m256i*>(_Last)), 78);
    const __m256i _Left_reversed  = _mm256_shuffle_epi8(_Left,
                                                        _Reverse_short_lanes_avx);
    const __m256i _Right_reversed = _mm256_shuffle_epi8(_Right,
                                                        _Reverse_short_lanes_avx);
    _mm256_storeu_si256(static_cast<__m256i*>(_First), _Right_reversed);
    _mm256_storeu_si256(static_cast<__m256i*>(_Last), _Left_reversed);
    _Advance_bytes(_First, 32);
  } while (_First != _Stop_at);
  ...
}
```

std::sort

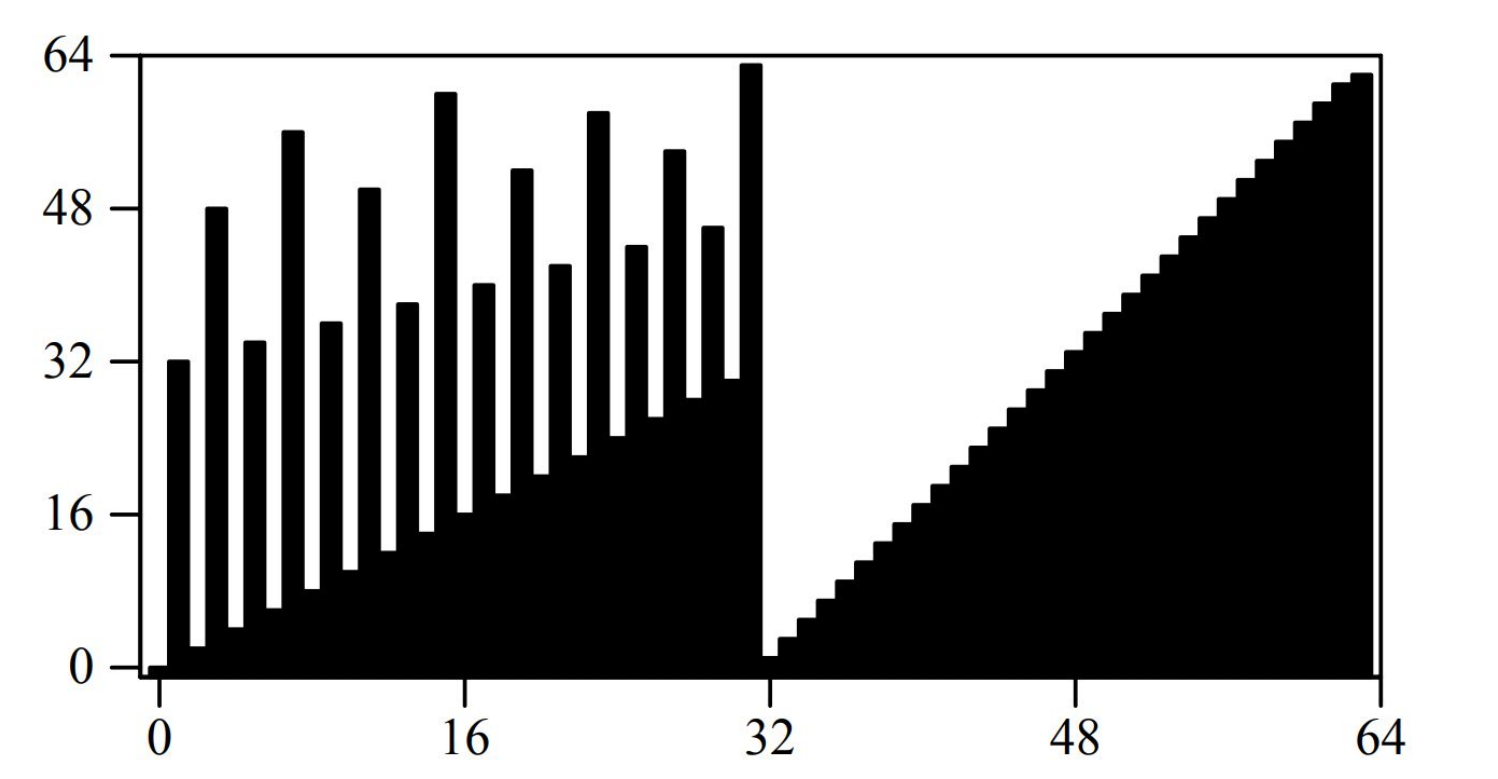
- Must have O(n log n) comparisons
- People debate about the best algorithms
  - pdqsort
  - Introsort
  - countsort
  - etc

std::sort

- libc++ has quadratic sort
  - qsort with some tricks

https://bugs.llvm.org/show_bug.cgi?id=20837

std::sort



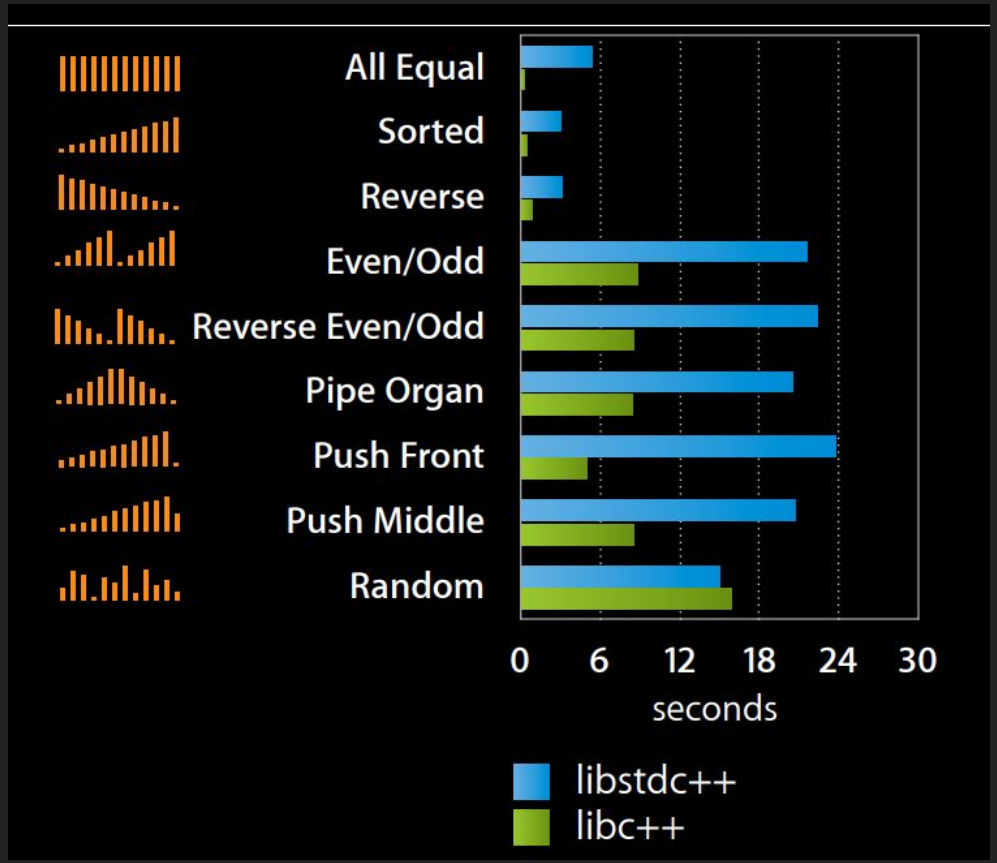
https://www.cs.dartmouth.edu/~doug/mdmspe.pdf

`std::sort`

n = 1000

libc++	251232 comparisons
libstdc++	29023 comparisons

std::sort


All Equal
Sorted
Reverse
Even/Odd
Reverse Even/Odd
Pipe Organ
Push Front
Push Middle
Random
0
6
12
18
24
30
seconds
libstdc++
libc++

`std::minmax_element`

- Uses 3/2n + O(1) comparisons
  - `min_element + max_element` are 2n comparisons
- For trivial types can be worse

std::minmax_element

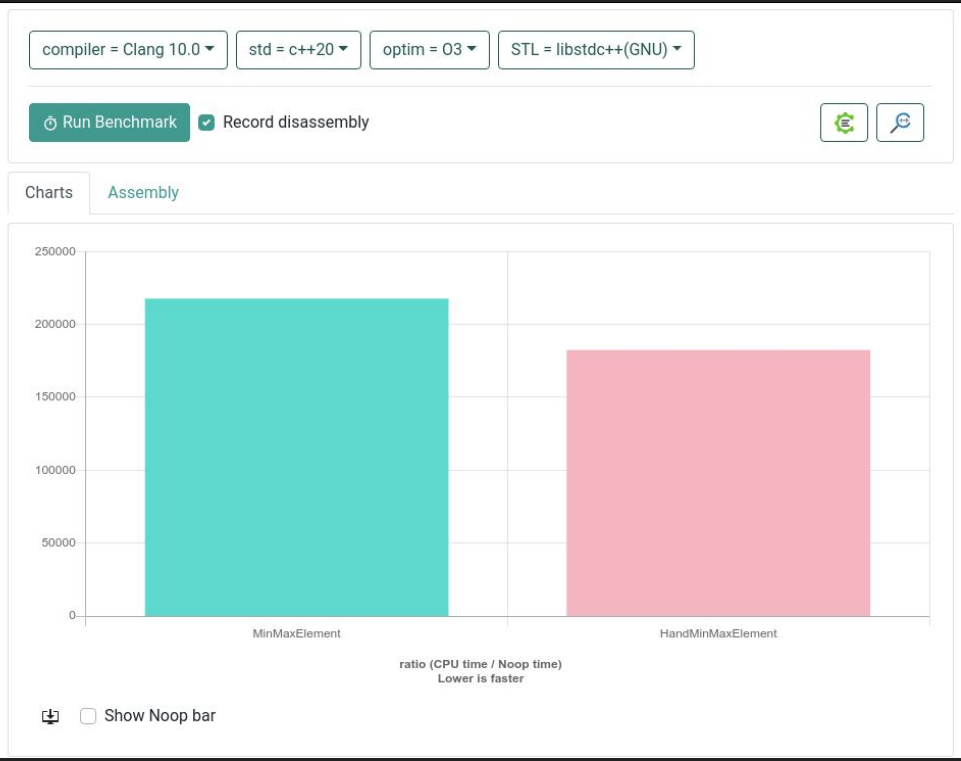
```cpp
#include <random>
#include <vector>
#include <algorithm>

std::vector<int> generate_data(size_t size) {
    using value_type = int;
    std::uniform_int_distribution<value_type> distribution(
        std::numeric_limits<value_type>::min(),
        std::numeric_limits<value_type>::max());
    std::default_random_engine generator;

    std::vector<value_type> data(size);
    std::generate(data.begin(), data.end(), [&]() { return distribution(generator); });
    return data;
}

static void MinMaxElement(benchmark::State& state) {
  auto v = generate_data(100000);
  for (auto _ : state) {
    auto [minimum, maximum] = std::minmax_element(v.begin(), v.end());
    benchmark::DoNotOptimize(minimum);
    benchmark::DoNotOptimize(maximum);
  }
}
BENCHMARK(MinMaxElement);

static void HandMinMaxElement(benchmark::State& state) {
  auto v = generate_data(100000);
  for (auto _ : state) {
    auto minimum = std::min_element(v.begin(), v.end());
    auto maximum = std::max_element(v.begin(), v.end());
    benchmark::DoNotOptimize(minimum);
    benchmark::DoNotOptimize(maximum);
  }
}
BENCHMARK(HandMinMaxElement);
```



Trick #7

Use standard algorithms, they are almost always good and they are constantly improved

<atomic>

- “Happens before” memory model
- Supported everywhere
  - x86-64, ARM, PowerPC, etc
    - 16 byte atomics!
    - More than 16 is not supported almost anywhere
  - CUDA (finally!)
- `volatile` is deprecated

<atomic>

CppCon 2019: JF Bastien “Deprecating volatile”

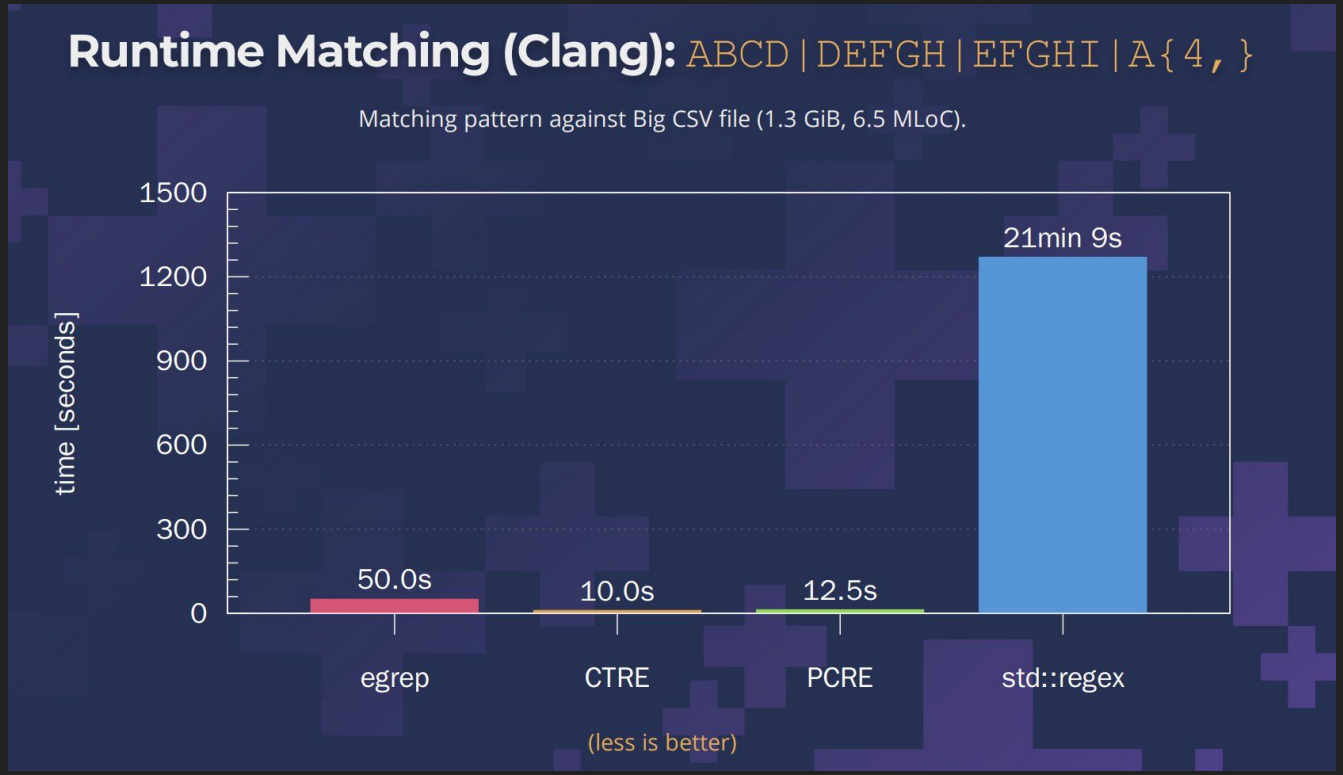
Trick #8

Use atomics, they are sane

std::regex

- It must support different grammars
  - BRE, ERE, ECMAscript, grep, egrep, awk, sed, etc.
  - It becomes a part of ABI

std::regex



Hana Dusikova, Compile Time Regular Expressions, CppCon 2018

`std::regex`

- `std::regex` crashes when matching long lines
  - 2 years
- C++11 `std::regex` memory corruption
  - 6 years
- C++11 `std::regex` resource exhaustion
  - 6 years

Trick #9

Never use `std::regex`

Trick #9

RE2, Hyperscan, PCRE, CTRE, boost::regex are much better

Real world

- Balance between speed and readability
  - Education, people onboarding
  - Last 1% might be more expensive in a long run
    - Debugging, occasional bugs

Real world. libc++ vs libstdc++

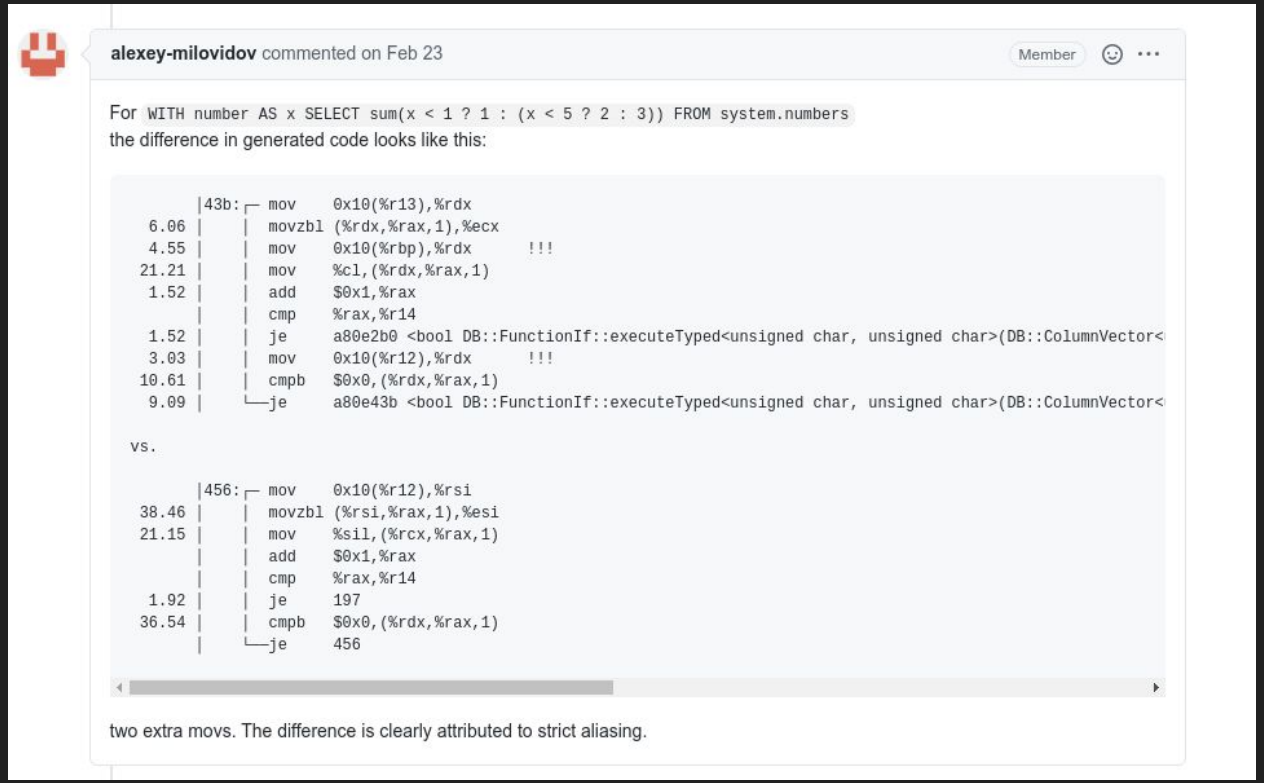
- Once ClickHouse decided to update the standard library from libstdc++ to libc++

https://github.com/ClickHouse/ClickHouse/pull/8311

Real world. C++17-20

- C++17-C++20
- char8_t
- 2% win

- C++14-C++17
- noexcept
- Copy elision
- 99 quantile win



alexey-milovidov commented on Feb 23 — Member

For `WITH number AS x SELECT sum(x < 1 ? 1 : (x < 5 ? 2 : 3)) FROM system.numbers`
the difference in generated code looks like this:

```
       |43b:┌─ mov    0x10(%r13),%rdx
  6.06 |    |  movzbl (%rdx,%rax,1),%ecx
  4.55 |    |  mov    0x10(%rbp),%rdx      !!!
 21.21 |    |  mov    %cl,(%rdx,%rax,1)
  1.52 |    |  add    $0x1,%rax
       |    |  cmp    %rax,%r14
  1.52 |    |  je     a80e2b0 <bool DB::FunctionIf::executeTyped<unsigned char, unsigned char>(DB::ColumnVector<
  3.03 |    |  mov    0x10(%r12),%rdx      !!!
 10.61 |    |  cmpb   $0x0,(%rdx,%rax,1)
  9.09 |    └─je     a80e43b <bool DB::FunctionIf::executeTyped<unsigned char, unsigned char>(DB::ColumnVector<

vs.

       |456:┌─ mov    0x10(%r12),%rsi
 38.46 |    |  movzbl (%rsi,%rax,1),%esi
 21.15 |    |  mov    %sil,(%rcx,%rax,1)
       |    |  add    $0x1,%rax
       |    |  cmp    %rax,%r14
  1.92 |    |  je     197
 36.54 |    |  cmpb   $0x0,(%rdx,%rax,1)
       |    └─je     456
```

two extra movs. The difference is clearly attributed to strict aliasing.

Trick #10

Write benchmarks, try different things, find your own best

ABI breakages

- P2028 paper about ABI future
  - Prague meeting results:
    - Committee can consider ABI breakage proposals
    - Only for huge performance wins
    - Do not break much
    - Be loud about the decision(?)

Trick #11

C++ is more than performance

Questions?